

Machine Learning Homework 7 Report

Kernel Eigenfaces, t-SNE and symmetric SNE

312554056 紀品榕

I. Code with detailed explanations

A. Kernel Eigenfaces

a. PCA algorithms overflow

The perform_PCA function in Figure 1 first calculates the projection matrix W from the covariance matrix of training data for dimension reduction. Then, W is displayed as eigenfaces, ten randomly picked trained images are reconstructed, and the training and testing data are projected to lower dimensions. KNN of the $k=5$ algorithm is used to classify the subject class of the testing data, and the face recognition performance is also calculated. Each steps are described below.

```
def perform_PCA(kernelmode,train_image,train_label,test_image,test_label,rows,cols,gamma,degree,k):
    # matrix for eigen decomposition
    kernel_list=["linear","polynomial","RBF"]
    title = "PCA"
    if kernelmode != 0:
        title = str(kernel_list[kernelmode-1])+" kernel "+title

    kernel = calculate_kernel(kernelmode,train_image,rows,cols,gamma,degree)
    if kernelmode == 0:
        matrix = kernel
    else:
        one_N = np.ones((rows*cols,rows*cols),dtype=float)/(rows*cols)
        matrix = kernel-one_N.dot(kernel)-kernel.dot(one_N)+one_N.dot(kernel).dot(one_N)

    # get projection matrix and show reconstruction
    largest_eigenvectors = get_largest_eigenvectors(matrix)
    show_eigenface_fisherface(title,largest_eigenvectors,"eigenface",rows,cols)
    choices, reconstructions = reconstruct_face(train_image,rows,cols,largest_eigenvectors)
    show_reconstruction_face(title,train_image,choices,reconstructions,rows, cols)

    # project data to low-d space
    low_dim_train = train_image@largest_eigenvectors
    low_dim_test = test_image@largest_eigenvectors

    # face recognition
    prediction = perform_knn_prediction(low_dim_train,train_label,low_dim_test,k)
    calculate_performance(title, prediction,test_label)

    return
```

(Figure 1)

b. Kernel PCA algorithms overflow

The perform_PCA function in Figure @ also performs linear, polynomial, and RBF kernel PCA calculations. After getting the kernel result, the matrix K^C for projection matrix W computation is calculated by the function below,

$$K^C = K - 1_N K - K 1_N + 1_N K 1_N$$

where K is the kernel result, N is the resized image pixel number and the size of the kernel, and 1_N is an $N * N$ matrix filled with element $1/N$.

The steps of calculating eigenfaces, reconstructing faces, projecting data to lower dimensions, performing KNN of the $k=5$ for classification, and calculating the face recognition performance are the same as those in PCA. Each step is described below.

c. Kernel or covariance matrix calculation

The *kernelmode* parameter determines whether to perform PCA or kernel PCA. The *calculate_kernel* function in Figure 2 computes the covariance matrix of the training data for further eigenvector computing if the PCA is performed, and the kernel result of the training data is calculated if the kernel PCA is performed.

```
def calculate_kernel(mode,image,rows,cols,gamma,degree):
    if mode == 0: # without kernel
        kernel = np.cov(image.T,bias=True)
    elif mode == 1: # linear kernel
        kernel = image.T.dot(image)
    elif mode == 2: # polynomial kernel
        kernel = np.power((image.T.dot(image)+gamma),degree)
    else: # RBF kernel
        kernel = np.exp(-gamma*cdist(image.T,image.T,'seuclidean'))
    return kernel
```

(Figure 2)

d. Eigenface construction and display

The *get_largest_eigenvectors* function in Figure 3 calculates the eigenvectors and gets the eigenvectors with the 25 largest eigenvalues as the projection matrix *W*. Then, the *show_eigenface_fisherface* function in Figure 4 displays the eigenface from the projection matrix.

```
def get_largest_eigenvectors(kernel):
    eigenvalues, eigenvectors = np.linalg.eig(kernel)
    largest_idx = np.argsort(-eigenvalues)[:25]
    largest_eigenvectors = eigenvectors[:, largest_idx].real
    return largest_eigenvectors
```

(Figure 3)

```
def show_eigenface_fisherface(title,largest_eigenvectors,face_type,rows,cols):
    plt.figure()
    plt.suptitle(face_type)
    for idx in range(25):
        plt.subplot(5,5,idx+1)
        plt.axis('off')
        plt.imshow(largest_eigenvectors.T[idx,:].reshape((rows,cols)), cmap='gray')
    plt.savefig(title+face_type+'.png')
    return
```

(Figure 4)

e. Images reconstruction and display

The *reconstruct_face* in Figure 5 function randomly picks ten images to

reconstruct, and each image reconstruction is done by the function below,

$$xWW^T$$

where x is the original image, W is the projection matrix, xW is the coordinates in the lower dimensional space, and xWW^T is the attempt to reconstruct the original image.

The *show_reconstruction_face* in Figure 6 function displays the original and the corresponding reconstruction image.

```
def reconstruct_face(train_image,rows,cols,projection):
    reconstructions = np.zeros((10,rows*cols))
    choices = np.random.choice(train_image.shape[0],10)
    for idx in range(10):
        reconstructions[idx,:] = train_image[choices[idx],:](projection)@(projection.T)
    return choices, reconstructions
```

(Figure 5)

```
def show_reconstruction_face(title,train_image,choices,reconstructions,rows, cols):
    fig = plt.figure()
    fig.suptitle('Original faces & Reconstructed faces')
    for idx in range(10):
        plt.subplot(2,10,idx*2+1)
        plt.axis('off')
        plt.imshow(train_image[choices[idx],:].reshape((rows,cols)),plt.cm.gray)

        plt.subplot(2,10,idx*2+2)
        plt.axis('off')
        plt.imshow(reconstructions[idx,:].reshape((rows,cols)),plt.cm.gray)

    plt.savefig(title+" reconstruction.png")
    return
```

(Figure 6)

f. Face recognition and performance computation

With training and testing data in lower-dimension space, the KNN algorithm is performed by the *perform_knn_prediction* function in Figure 7 to classify which subject each testing image belongs to. Then, the prediction result is compared with the truth image label for performance calculation in the *calculate_performance* function in Figure 8.

```
def perform_knn_prediction(low_dim_train,train_label,low_dim_test,k):
    train_num = low_dim_train.shape[0]
    test_num = low_dim_test.shape[0]
    prediction = np.zeros(test_num,dtype=int)

    for test_idx in range(test_num):
        dist = np.zeros(train_num)
        for train_idx in range(train_num):
            dist[train_idx] = np.linalg.norm(low_dim_test[test_idx]-low_dim_train[train_idx])
        k_nearest = train_label[np.argsort(dist)[:k]]
        prediction[test_idx] = np.argmax(np.bincount(k_nearest))

    return prediction
```

(Figure 7)

```
def calculate_performance(title, prediction, test_label):
    error = 0.0
    test_num = test_label.shape[0]
    for idx in range(test_num):
        if prediction[idx] != test_label[idx]:
            error += 1
    print(title)
    print("Error rate: ", error/test_num, "(" + str(int(error)) + "/" + str(test_num) + ")")
    return
```

(Figure 8)

g. LDA algorithms overflow

The perform_LDA function in Figure 9 performs the LDA algorithm. It first calculates between-class scatter S_B and within-class scatter S_W of the original data points in the *get_LDA_matrix* function in Figure 10 as below.

$$S_W = \sum_{j=1}^k S_j = \sum_{j \in C_j} (x_i - m_j)(x_i - m_j)^T, \text{ where } m_j = \frac{1}{n_j} \sum_{j \in C_j} x_i$$

$$S_B = \sum_{j=1}^k S_{B_j} = \sum_{j=1}^k n_j (m_j - m)(m_j - m)^T, \text{ where } m = \frac{1}{n} \sum x$$

where x_i is the data point, n_j is the count of each class, m_j is the mean of each subject type, and m is the mean of all the data points.

Then, get the first 25 largest eigenvectors of the function below as the projection matrix W .

$$W = \text{first 25 largest eigenvectors of } S_W^{-1} S_B$$

The projection matrix is used to construct fisherfaces. Other steps, like projecting data to lower dimensions, performing KNN of the $k=5$ algorithm for classification, and calculating the face recognition performance, are the same as those in PCA.

```

def perform_LDA(kernelmode,train_image,train_label,test_image,test_label,rows,cols,gamma,degree,k):
    kernel_list=["linear","polynomial","RBF"]
    title = "LDA"

    if kernelmode != 0:
        title = str(kernel_list[kernelmode-1])+" kernel "+title

    if kernelmode ==0:
        matrix = get_LDA_matrix(kernelmode,train_image,train_label,test_image,test_label,rows,cols,gamma,degree,k)
    else:
        matrix = get_kernel_LDA_matrix(kernelmode,train_image,train_label,test_image,test_label,rows,cols,gamma,degree,k)

    # get projection matrix and show reconstruction
    largest_eigenvectors = get_largest_eigenvectors(matrix)
    show_eigenface_fisherface(title, largest_eigenvectors,"fisherface",rows,cols)
    choices, reconstructions = reconstruct_face(train_image,rows,cols,largest_eigenvectors)
    show_reconstruction_face(title, train_image,choices,reconstructions,rows, cols)

    # project data to low-d space
    low_dim_train = train_image@largest_eigenvectors
    low_dim_test = test_image@largest_eigenvectors

    # face recognition
    prediction = perform_knn_prediction(low_dim_train,train_label,low_dim_test,k)
    calculate_performance(title,prediction,test_label)
    print("-"*50)

    return

```

(Figure 9)

```

def get_LDA_matrix(kernelmode,train_image,train_label,test_image,test_label,rows,cols,gamma,degree,k):
    labels, repeats = np.unique(train_label,return_counts=True)
    subject_num = len(labels)

    kernel = None
    if kernelmode == 0:
        kernel = train_image
    else:
        kernel = calculate_kernel(kernelmode,train_image,rows,cols,gamma,degree)

    train_num = train_image.shape[0]
    pixel_num = kernel.shape[1]

    # get mean
    total_mean = np.mean(kernel,axis=0).reshape(-1, 1)
    subject_mean = np.zeros((subject_num, pixel_num))

    for i in range(train_num):
        for p in range(pixel_num):
            subject_mean[train_label[i]-1,p] += kernel[i,p]
    for idx in range(subject_num):
        subject_mean[idx] /= repeats[idx]

    # calculate similarities
    s_within = np.zeros((pixel_num,pixel_num))
    s_between = np.zeros((pixel_num,pixel_num))

    for idx in range(train_num):
        distance = np.array(kernel[idx]-subject_mean[train_label[idx]-1]).reshape(-1,1)
        s_within += distance@distance.T

    for idx in range(subject_num):
        distance = np.array(subject_mean[idx]-total_mean[idx]).reshape(-1,1)
        s_between += repeats[idx]*(distance@distance.T)

    return np.linalg.pinv(s_within)@s_between

```

(Figure 10)

h. Kernel LDA algorithms overflow

The *perform_LDA* function also performs the kernel LDA algorithm. In the *get_kernel_LDA_matrix* function in Figure 11, matrix N and M are calculated as below,

$$N = \sum_{j=1}^c K_j (I - 1_{l_j}) K_j^T$$

$$M = n_j \sum_{j=1}^c (M_j - M_*)(M_j - M_*)^T$$

$$(M_*)_j = \frac{1}{l} \sum_{k=1}^l k(x_j, x_k)$$

$$(M_i)_j = \frac{1}{n_j} \sum_{k=1}^{n_j} k(x_j, x_k^i)$$

where x_i is the data point, n_j is the count of each class, l is the count of an image, $k(x_j, x_k)$ is the kernel result, M_j represents the mean of each kernel of different subject type, and M_* represents the mean of the kernel.

```
def get_kernel_LDA_matrix(kernelmode,train_image,train_label,test_image,test_label,rows,cols,gamma,degree,k):
    labels, repeats = np.unique(train_label,return_counts=True)
    subject_num = len(labels)

    # calculate kernel results
    kernel = calculate_kernel(kernelmode,train_image,rows,cols,gamma,degree)

    pixel_num = kernel.shape[1]

    subject_kernel = np.zeros((subject_num,pixel_num,pixel_num))
    for idx in range(subject_num):
        subject_image = train_image[train_label==idx+1]
        subject_kernel[idx] = calculate_kernel(kernelmode,subject_image,rows,cols,gamma,degree)

    # get mean
    total_mean = np.mean(kernel,axis=0).reshape(-1, 1)
    subject_mean = np.zeros((subject_num, pixel_num))

    for idx in range(subject_num):
        subject_mean[idx] = np.sum(subject_kernel[idx]).reshape(-1, 1)/subject_num

    # calculate similarities
    M = np.zeros((pixel_num,pixel_num))
    N = np.zeros((pixel_num,pixel_num))

    for idx in range(subject_num):
        distance = repeats[idx]*np.array(subject_mean[idx]-total_mean).reshape(-1,1)
        M += distance.dot(distance.T)

    identity = np.eye(pixel_num)
    for idx in range(subject_num):
        one_li = np.ones((pixel_num,pixel_num))/repeats[idx]
        N += subject_kernel[idx].dot(identity-one_li).dot(subject_kernel[idx].T)

    return np.linalg.pinv(N)@M
```

(Figure 11)

Then, get the first 25 largest eigenvectors of the function below as the projection matrix W .

$$W = \text{first 25 largest eigenvectors of } N^{-1}M$$

The projection matrix is used to construct fisherfaces. Other steps, like projecting data to lower dimensions, performing the KNN algorithm of the $k=5$ for classification, and calculating the face recognition performance, are the same as

those in PCA.

i. Fisherface construction and display

The `get_largest_eigenvectors` function in Figure 3 calculates the eigenvectors and gets the eigenvectors with the 25 largest eigenvalues as the projection matrix W . Then, the `show_eigenface_fisherface` function in Figure 4 displays the fisherface from the projection matrix.

Moreover, some other implementations have been made for fisherface construction [1]. One of them is combining PCA and Fisher's Linear Discriminant (FLD). It includes modification of the PCA projection matrix and the projection matrix formula. The implementation and the results are shown in Section III. Observations and Discussion

B. t-SNE

a. modification from t-SNE to symmetric SNE

The difference between t-SNE and symmetric SNE lies in the similarity formula of data points in low dimensionality $q_{j|i}$, and the gradient $\frac{\partial C}{\partial y_i}$.

Modifications based on derivation are added to transform one algorithm from another.[2]

The similarity formula of data points in low dimensionality and the gradient of t-SNE is shown below,

$$q_{ij} = \frac{(1 + \|y_i - y_j\|^2)^{-1}}{\sum_{k \neq l} (1 + \|y_k - y_l\|^2)^{-1}}$$

$$\frac{\partial C}{\partial y_i} = 4 \sum_j (p_{ij} - q_{ij}) (1 + \|y_i - y_j\|^2)^{-1} (y_i - y_j)$$

where y_i is the low-dimensional representation of individual data points, p_{ij} is the similarity of data points in high-dimensionality, and C is the cost function.

The similarity formula of data points in low dimensionality and the gradient of symmetric SNE is shown below,

$$q_{ij} = \frac{\exp(-\|y_i - y_j\|^2)}{\sum_{k \neq l} \exp(-\|y_k - y_l\|^2)}$$

$$\frac{\partial C}{\partial y_i} = 4 \sum_j (p_{ij} - q_{ij}) (y_i - y_j)$$

The modification part of the code is shown in Figure 12. If the *mode* parameter is 0, t-SNE is performed; if the *mode* parameter is 1, symmetric SNE is performed.

```

if mode == 0:
    num = 1. / (1. + np.add(np.add(num, sum_Y).T, sum_Y))
else:
    num = np.exp(-1. * np.add(np.add(num, sum_Y).T, sum_Y))

for i in range(n):
    if mode == 0:
        dY[i, :] = np.sum(np.tile(PQ[:, i] * num[:, i], (no_dims, 1)).T * (Y[i, :] - Y), 0)
    else:
        dY[i, :] = np.sum(np.tile(PQ[:, i], (no_dims, 1)).T * (Y[i, :] - Y), 0)

```

(Figure 12)

b. visualization of the embedding

For every ten iterations, the data are projected onto 2D space marked by different colors on each label and saved by the *get_current_image* function in Figure 13. After finishing all the iterations, the GIF of the embedding process is created.

```

def get_current_image(Y, labels, mode, iter):
    x_lim=[[-100,100],[-10,10]]
    y_lim=[[-100,100],[-10,10]]
    plt.clf()
    plt.title("Iteration "+str(iter))
    plt.scatter(Y[:, 0], Y[:, 1], 5, labels)
    plt.xlim(x_lim[mode][0], x_lim[mode][1])
    plt.ylim(y_lim[mode][0], y_lim[mode][1])
    fig = plt.get_current_fig_manager().canvas
    fig.draw()
    img = Image.frombytes('RGB', fig.get_width_height(), fig.tostring_rgb())
    return img

```

(Figure 13)

c. visualization of the distribution of pairwise similarities

For the distribution of pairwise similarities in both high-dimensional and low-dimensional space of either t-SNE or symmetric, the *visualize_similarity* function in Figure 14 plots each similarity as a histogram with a log scale axis.


```
def visualize_similarity(P,Q,title):
    plt.clf()
    plt.title(title+" high-dimensional space")
    plt.hist(P.flatten(),bins=50,log=True)
    plt.savefig(title+"_high-d_space.png")

    plt.clf()
    plt.title(title+" low-dimensional space")
    plt.hist(Q.flatten(),bins=50,log=True)
    plt.savefig(title+"_low-d_space.png")

    return
```

(Figure 14)

d. choices of different perplexity values

Different choices of perplexity values are set and tested in the program.

```
if __name__ == "__main__":
    algo=["tSNE","sSNE"]
    perplexity = [10.0,20.0,30.0,40.0,50.0]
    x_lim=[[-100,100],[-10,10]]
    y_lim=[[-100,100],[-10,10]]
    print("Run V = tsne.tsne(X, no_dims, perplexity) to perform t-SNE on your dataset.")
    print("Running example on 2,500 MNIST digits...")
    X = np.loadtxt("mnist2500_X.txt")
    labels = np.loadtxt("mnist2500_labels.txt")
    for i in range(1,-1,-1):
        for p in range(len(perplexity)):
            start = time.time()
            title = algo[i]+"_perplexity_"+str(perplexity[p])
            plt.clf()
            V = perform_tsne_ssne(X, 2, 50, perplexity[p], i,title)

            plt.clf()
            plt.scatter(V[:, 0], V[:, 1], 5, labels)
            plt.xlim(x_lim[i][0],x_lim[i][1])
            plt.ylim(y_lim[i][0],y_lim[i][1])
            plt.savefig(title+".png")

        print("time: ",time.time()-start,"s.")
```

(Figure 15)

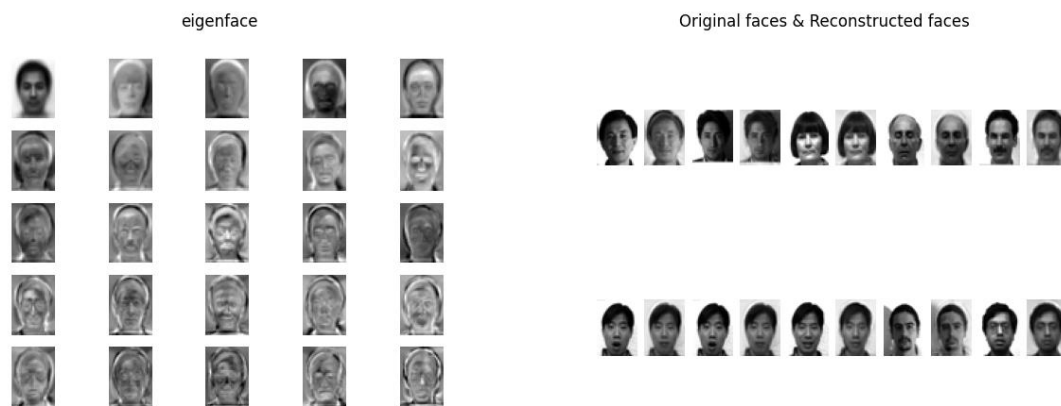
II. Experiments and Discussion

A. Kernel Eigenfaces

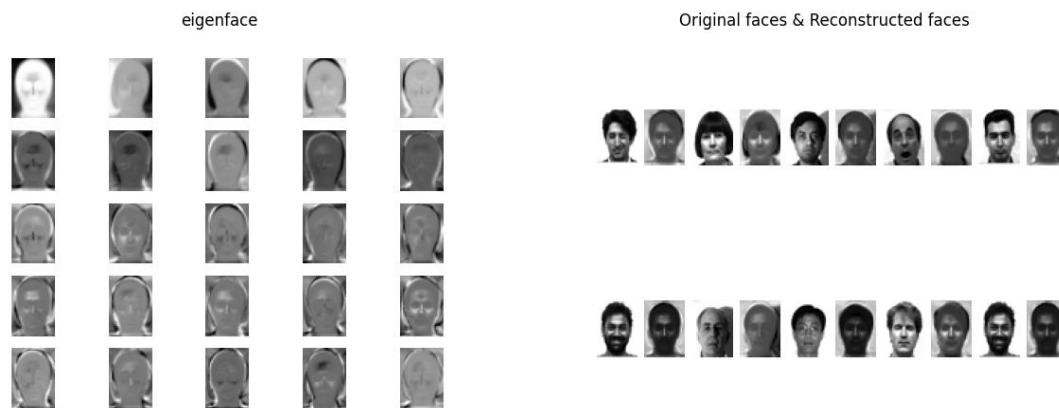
a. Eigenfaces and reconstructed faces of PCA and kernel PCA



(Figure 16. Eigenfaces and reconstructed faces of PCA)



(Figure 17. Eigenfaces and reconstructed faces of linear kernel PCA)



(Figure 18. Eigenfaces and reconstructed faces of polynomial kernel PCA)

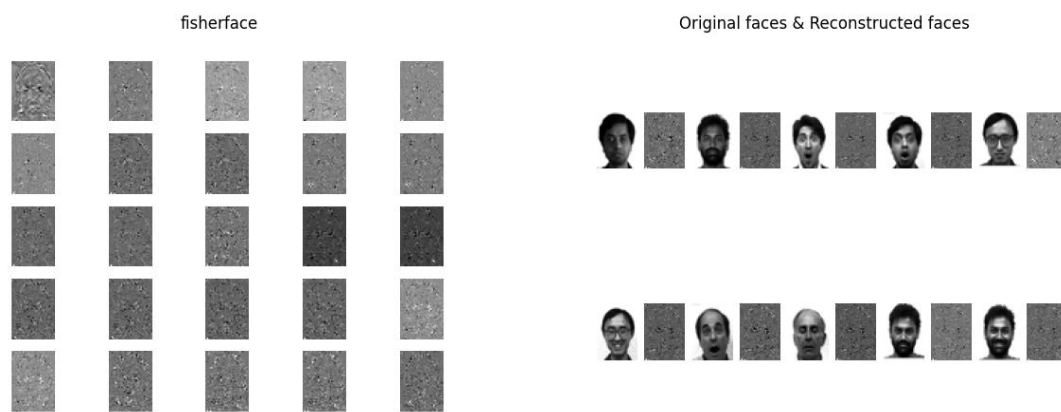


(Figure 19. Eigenfaces and reconstructed faces of RBF kernel PCA)

b. Fisherfaces and reconstructed faces of LDA and kernel LDA



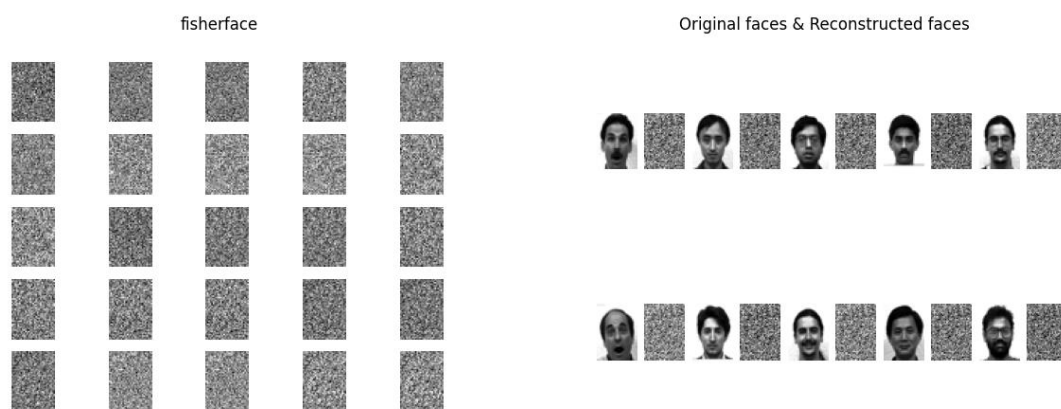
(Figure 20. Fisherfaces and reconstructed faces of LDA)



(Figure 21. Fisherfaces and reconstructed faces of linear kernel LDA)



(Figure 22. Fisherfaces and reconstructed faces of polynomial kernel LDA)



(Figure 23. Fisherfaces and reconstructed faces of RBF kernel LDA)

c. Face recognition performance

```
PS G:\我的雲端硬碟\cs12\machine_learning\HW7> python -\kernel_eigenface_nm.py
PCA
Error rate: 0.13333333333333333 ( 4 / 30 )
LDA
Error rate: 0.03333333333333333 ( 1 / 30 )
-----
linear kernel PCA
Error rate: 0.13333333333333333 ( 4 / 30 )
linear kernel LDA
Error rate: 0.2 ( 6 / 30 )
-----
polynomial kernel PCA
Error rate: 0.16666666666666666 ( 5 / 30 )
polynomial kernel LDA
Error rate: 0.26666666666666666 ( 8 / 30 )
-----
RBF kernel PCA
Error rate: 0.13333333333333333 ( 4 / 30 )
RBF kernel LDA
Error rate: 0.3 ( 9 / 30 )
```

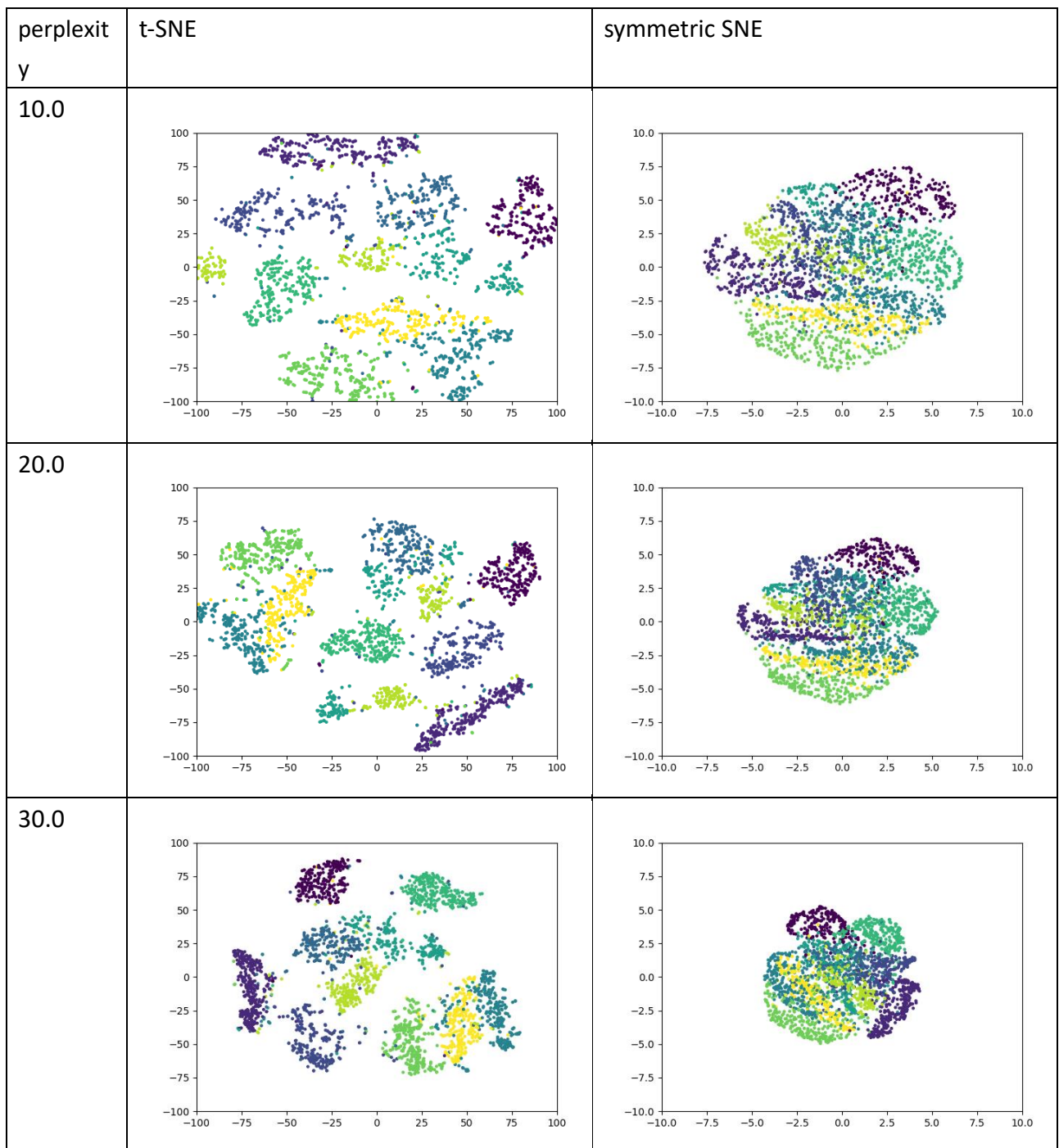
(Figure 24)

d. Observation

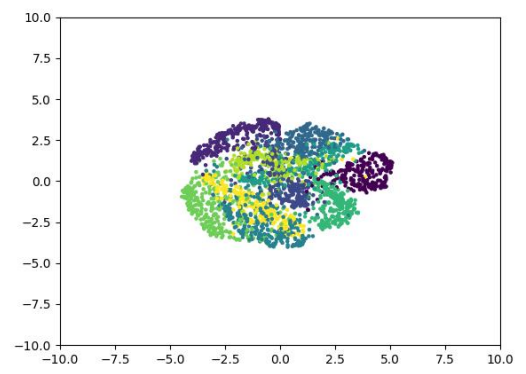
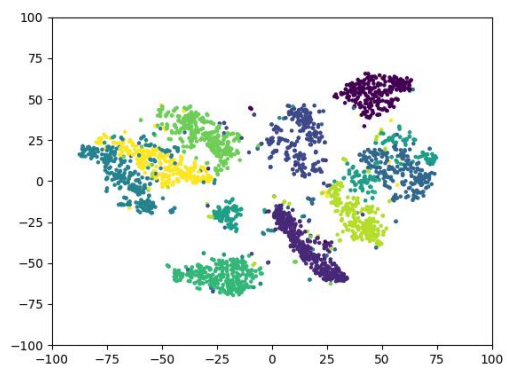
1. Compared with the face recognition results of PCA and LDA, PCA reaches a reasonable accuracy rate in most cases, while LDA only gets a better result in non-kernel LDA cases.
2. PCAs show more recognizable results for face reconstruction for human eyes.
 - Linear kernel PCA in Figure @ results in the most precise and most similar image, including the lightening of the image, the face structure, and the facial expression of the original image.
 - RBF kernel PCA in Figure @ also yields a clear and similar image to the original image. Still, some facial expressions in the reconstruction images differ from those in the original image.
 - PCA also reconstructs recognizable images, but the background of the reconstruction seems to be some noises, and the wrong facial expression problem also occurred.
 - Polynomial kernel PCA reconstructs recognizable but fuzzier facial expression images.
3. LDAs, whether LDAs or FDAs show unrecognizable reconstructions.
4. PCAs show more human-like eigenfaces, especially linear and RBF kernel RBF, while the others show invert-like eigenfaces. Also, eigenfaces with larger eigenvalues present more human-like faces.
5. LDAs, whether LDAs or FDAs show fuzzy looking fisherfaces. There might be something wrong with my implementation.

B. t-SNE

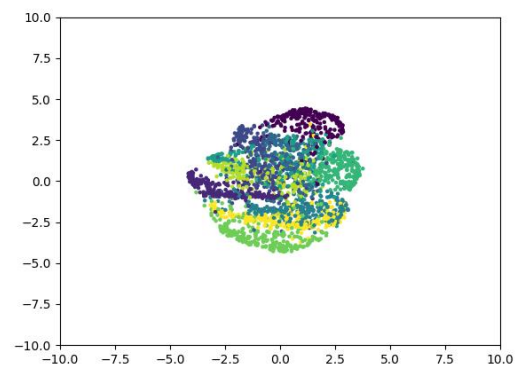
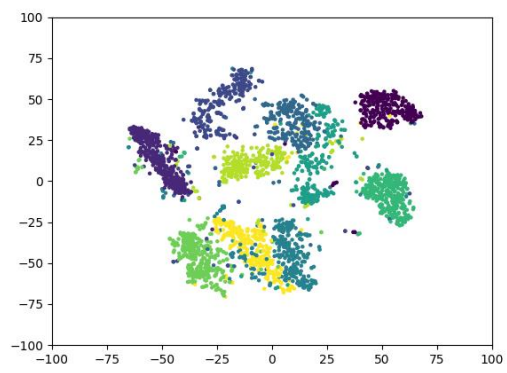
a. The embedding of both t-SNE and symmetric SNE after 1000 iterations



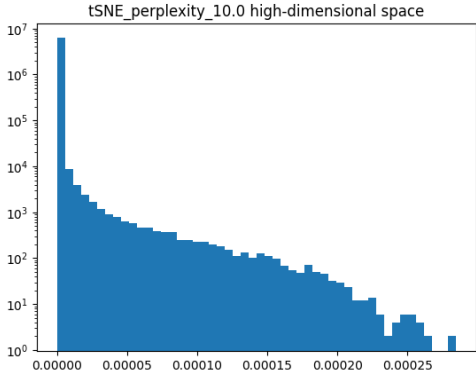
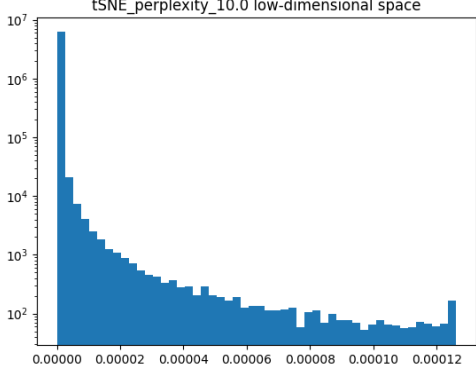
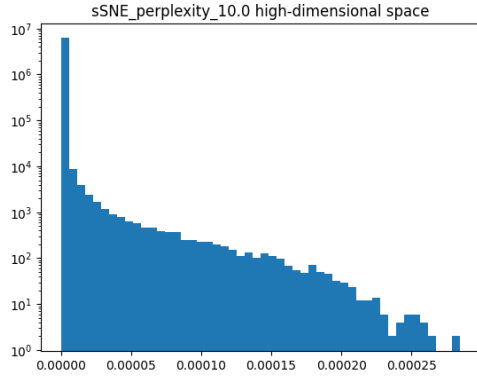
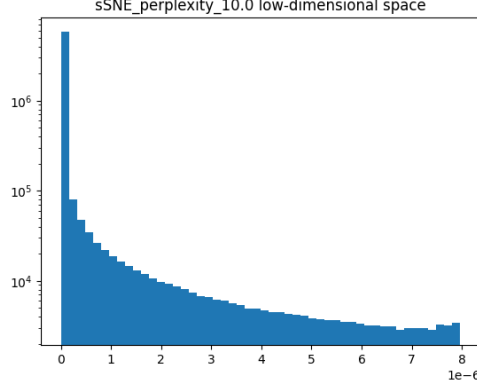
40.0



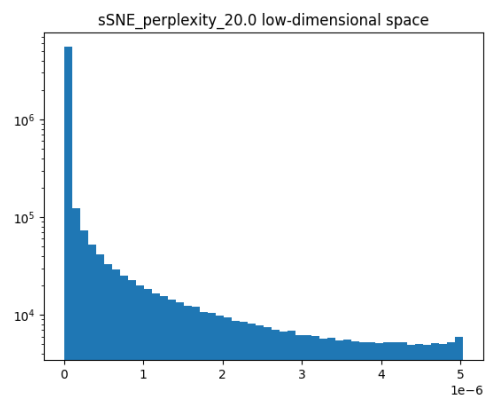
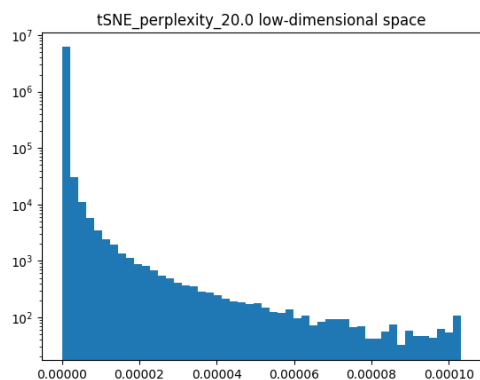
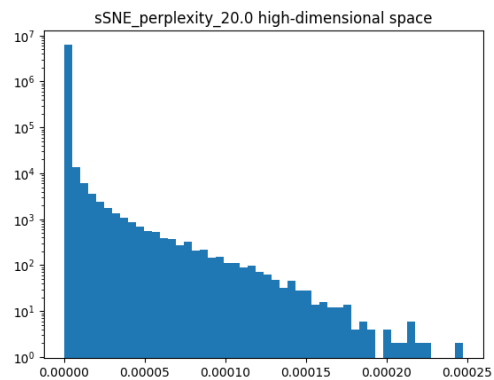
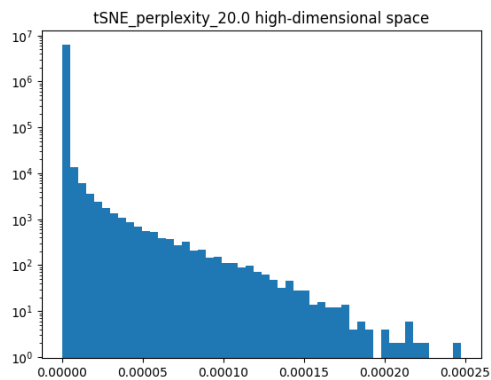
50.0



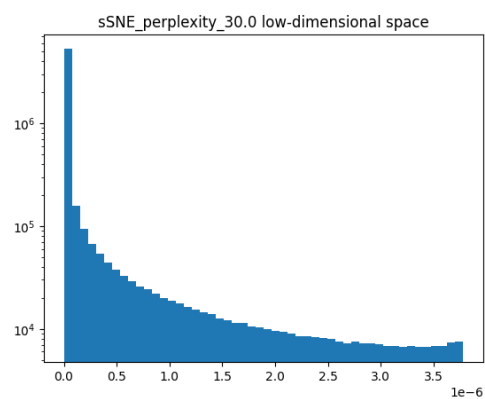
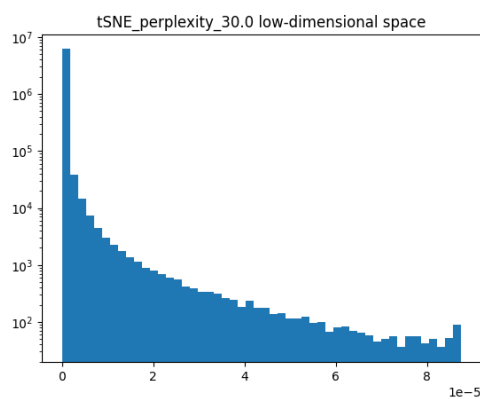
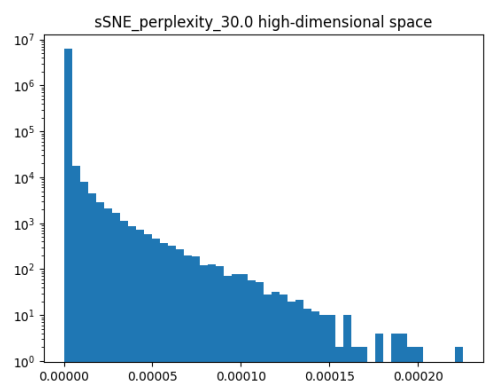
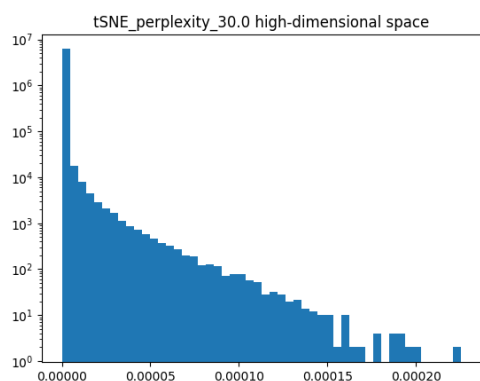
b. The distribution of pairwise similarities

perplexity	t-SNE	symmetric SNE
10.0	 <p>tSNE_perplexity_10.0 high-dimensional space</p>  <p>tSNE_perplexity_10.0 low-dimensional space</p>	 <p>sSNE_perplexity_10.0 high-dimensional space</p>  <p>sSNE_perplexity_10.0 low-dimensional space</p>

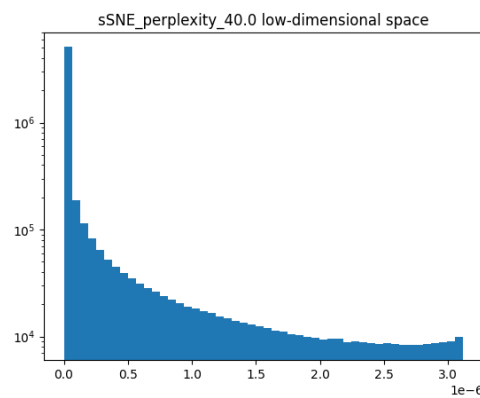
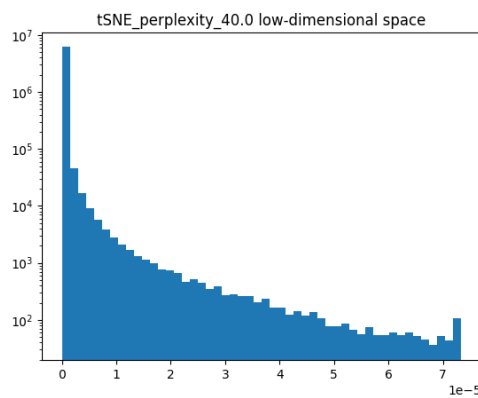
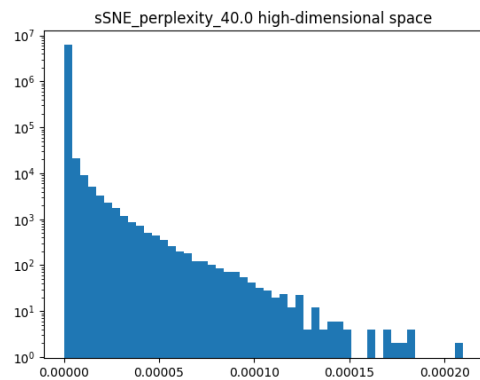
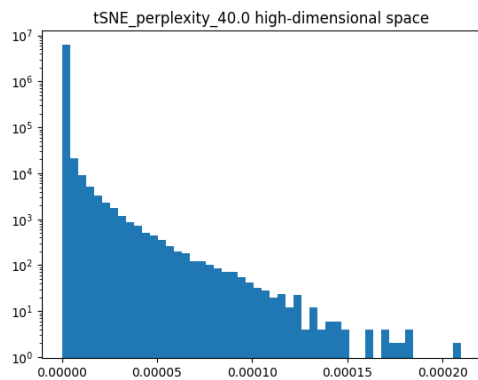
20.0



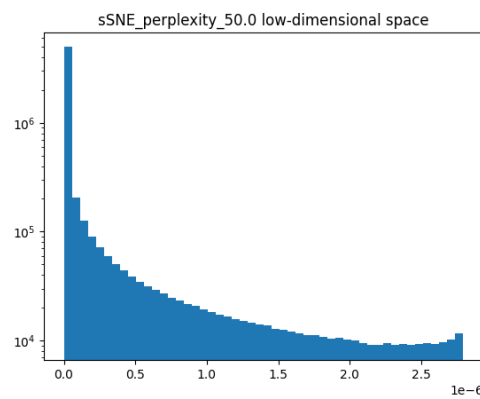
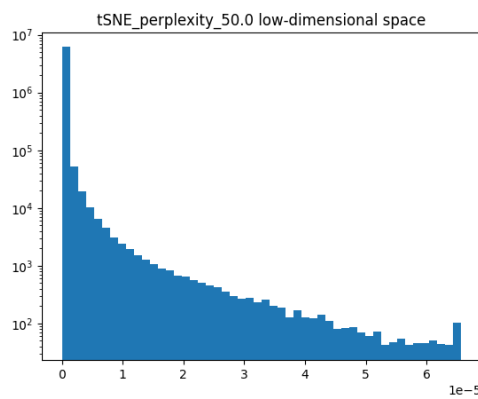
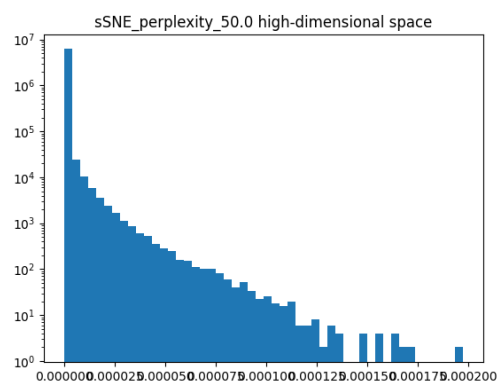
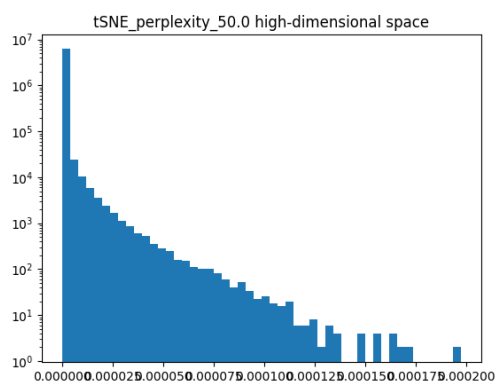
30.0



40.0



50.0



c. Observations

1. For the embedding results of the same perplexity, t-SNE separates the data points more widely; the axis range can easily be observed. Symmetric SNE suffers from the crowding problem since the data points still overlap even when the images are scaled up compared with t-SNE.
2. Perplexity can be thought of as choosing whether to preserve the data's global or local structure and thus affect the adequate number of neighbors.
 - For symmetric SNE, the crowding problem is more noticeable, and the influence of different perplexities is not apparent.
 - For tSNE, as perplexity increases, different clusters tend to shrink into denser structures, which probably shows that more global structure is retained than the local structure.

III. Observations and Discussion

A. Meanings of the eigenfaces

Eigenfaces construct a lower-dimensional subspace to represent most of the images in the training data. Eigenfaces seem to be the pattern or feature of human faces and record the lighting, face structure, facial expressions, or other facial characteristics of humans.

B. Implementation and the results of FKD

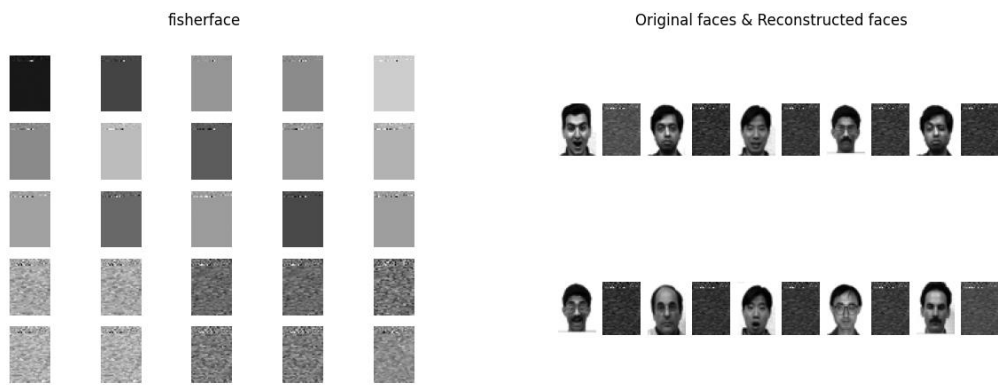
The implementation of FKD includes the calculation of the projection matrix of PCA and FLD, where the projection matrix of PCA W_{PCA} and the projection matrix of FLD W_{FLD} as below,

$$W_{PCA} = \operatorname{argmax}_W |W^T S_T W|, \text{ where } S_T = \sum_{k=1}^l (x_k - \mu)(x_k - \mu)^T$$
$$W_{FLD} = \operatorname{argmax}_W \frac{|W^T W_{PCA}^T S_B W W_{PCA}|}{|W^T W_{PCA}^T S_W W W_{PCA}|}$$

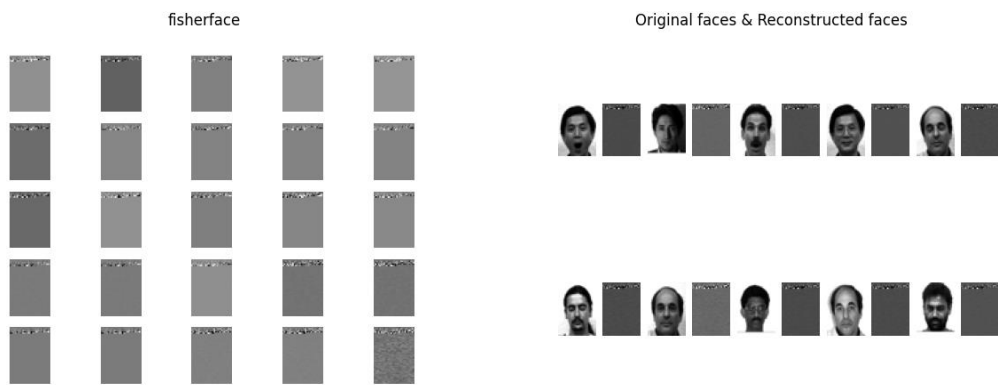
The projection matrix of FKD W_{FKD} is calculated as below,

$$W_{FKD}^T = \text{first 25 largest eigenvectors of } W_{FLD}^T W_{PCA}^T$$

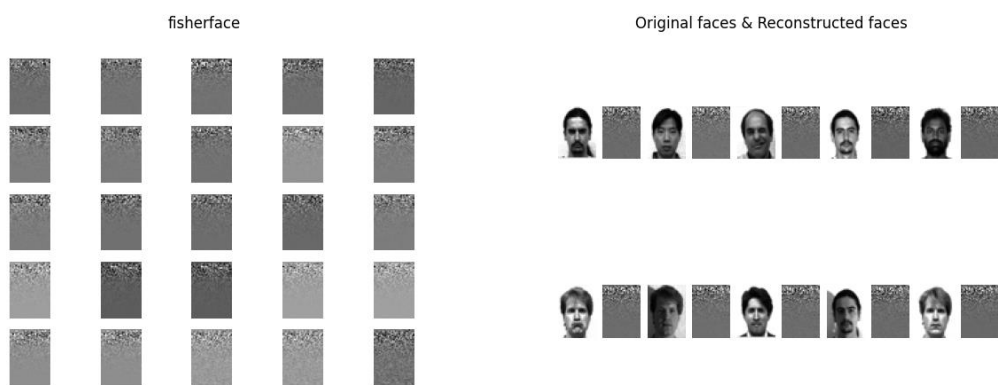
The remaining parts from fisherfaces displayed, as well as face reconstruction and recognition, are the same as LDA.



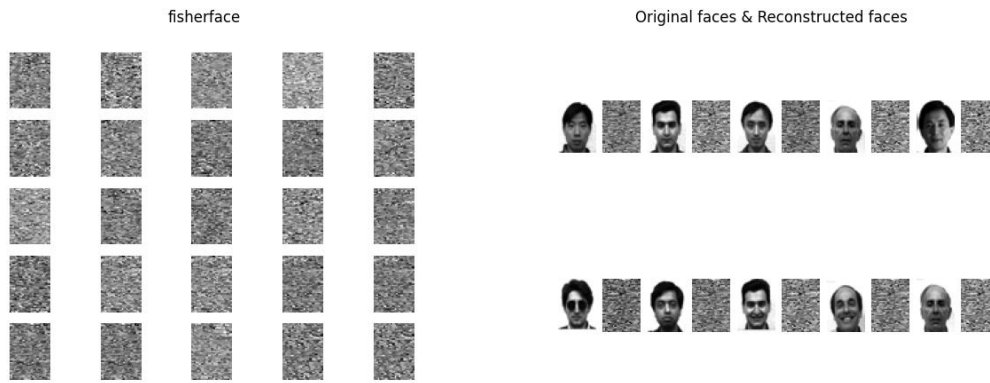
(Figure 25. Fisherfaces and reconstructed faces of FDA)



(Figure 26. Fisherfaces and reconstructed faces of linear kernel FLD)



(Figure 27. Fisherfaces and reconstructed faces of polynomial kernel FLD)



(Figure. Fisherfaces and reconstructed faces of RBF kernel FLD)

```
PS G:\我的雲端硬碟\cs12\machine_learning\HW7> python .\kernel_eigenface_fisher.py
PCA
Error rate: 0.13333333333333333 ( 4 / 30 )
LDA
Error rate: 0.3333333333333333 ( 10 / 30 )
-----
linear kernel PCA
Error rate: 0.13333333333333333 ( 4 / 30 )
linear kernel LDA
Error rate: 0.43333333333333335 ( 13 / 30 )
-----
polynomial kernel PCA
Error rate: 0.2 ( 6 / 30 )
polynomial kernel LDA
Error rate: 0.3 ( 9 / 30 )
-----
RBF kernel PCA
Error rate: 0.2 ( 6 / 30 )
RBF kernel LDA
Error rate: 0.13333333333333333 ( 4 / 30 )
```

(Figure 28. Face recognition performance of FLD)

IV. References

- [1] P. N. Belhumeur, J. P. Hespanha, and D. J. Kriegman, "Eigenfaces vs. Fisherfaces: Recognition Using Class Specific Linear Projection," *IEEE Trans. PATTERN Anal. Mach. Intell.*, vol. 19, no. 7, 1997.
- [2] L. van der Maaten and G. Hinton, "Visualizing Data using t-SNE," *J. Mach. Learn. Res.*, vol. 9, no. 86, pp. 2579–2605, 2008.